



April 2022,  
Course Java

# Object Oriented Programming in Java

Trayan Iliev

[tiliev@iproduct.org](mailto:tiliev@iproduct.org)

<http://iproduct.org>

Copyright © 2003-2022 IPT - Intellectual  
Products & Technologies

# About me



## Trayan Iliev

- CEO of IPT – Intellectual Products & Technologies
- Oracle® certified programmer 15+ Y
- end-to-end reactive fullstack apps with Java, ES6/7, TypeScript, Angular, React and Vue.js
- 12+ years IT trainer
- Voxxed Days, jPrime, jProfessionals, BGOUG, BGJUG, DEV.BG speaker
- Organizer RoboLearn hackathons and IoT enthusiast (<http://robolearn.org>)

# Where to Find the Code?

Java Web Development projects and examples are available @ GitHub:

<https://github.com/iproduct/java-fundamentals-2022>

# Agenda for This Session

- ❖ OOP principles – Encapsulation, Inheritance and Polymorphism, Overriding / Overloading
- ❖ String Processing,
- ❖ Data Formatting, Resource Bundles, Regular Expressions
- ❖ java.util & java.math
- ❖ StringTokenizer, Date/Calendar,
- ❖ Locale, Random, Optional, Observable, Observable interface, BigDecimal

# Basic Concepts in OOP and OOAD

- ❖ interface and implementation – we divide what remains constant (contractual interface) from what we would like to keep our freedom to change (hidden realization of this interface)
- ❖ interface = **public**
- ❖ implementation = **private**
- ❖ This separation allows the system to evolve while maintaining backward compatibility to already implemented solutions, enables parallel development of multiple teams
- ❖ **programming based on contractual interfaces**



# Object-Oriented Approach to Programming

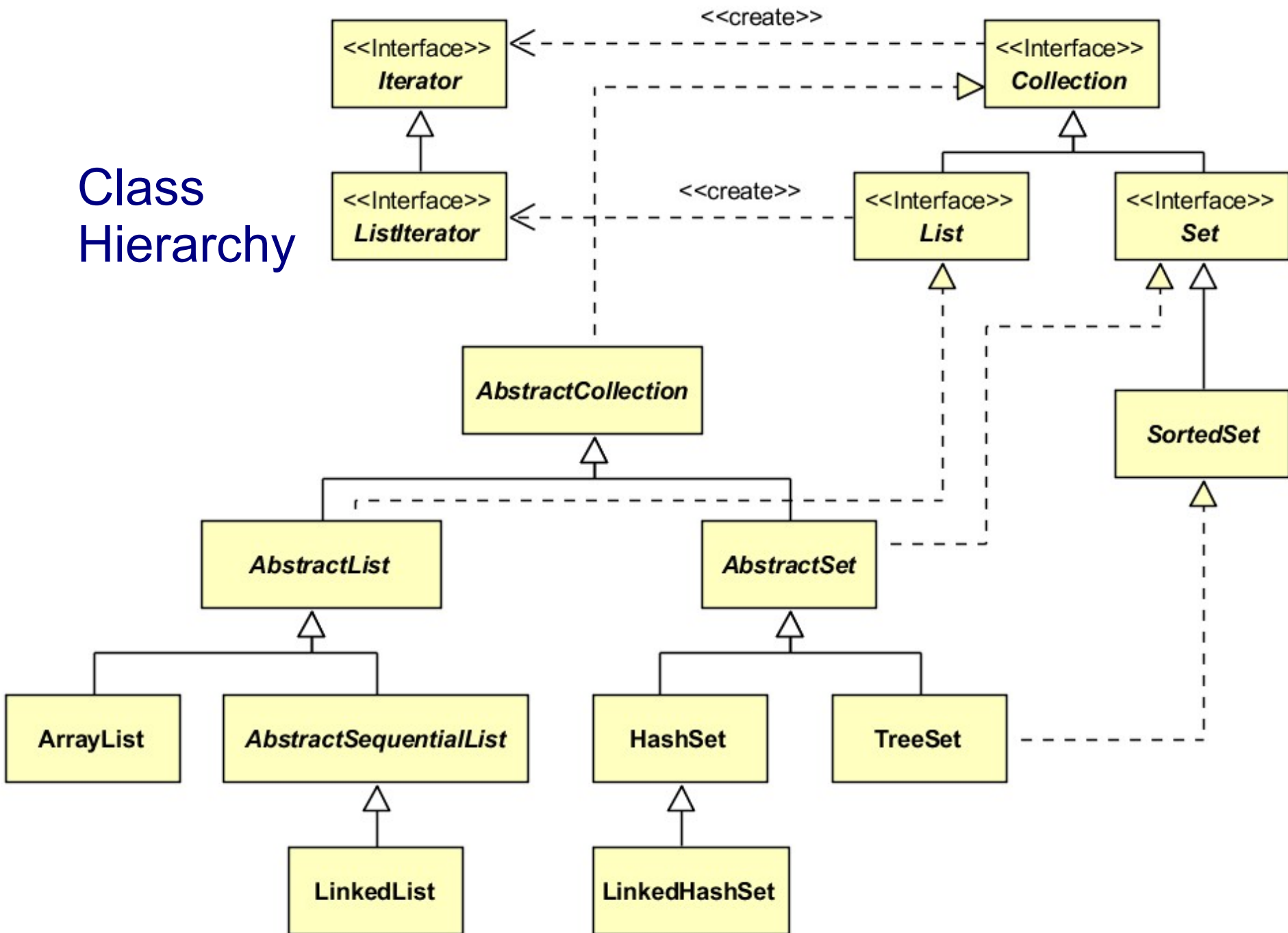
Key elements of the object model [Booch]:

- ❖ **class, object, interface and implementation**
- ❖ **abstraction** – basic distinguishing characteristics of an object
- ❖ **capsulation** – separating the elements of abstraction that make up its structure and behavior - interface and implementation
- ❖ **modularity** – decomposing the system into a plurality of components and loosely connected modules - principle: maximum coherence and the minimum connectivity
- ❖ **hierarchy** – class and object hierarchies

# SOLID Design Principles of OOP

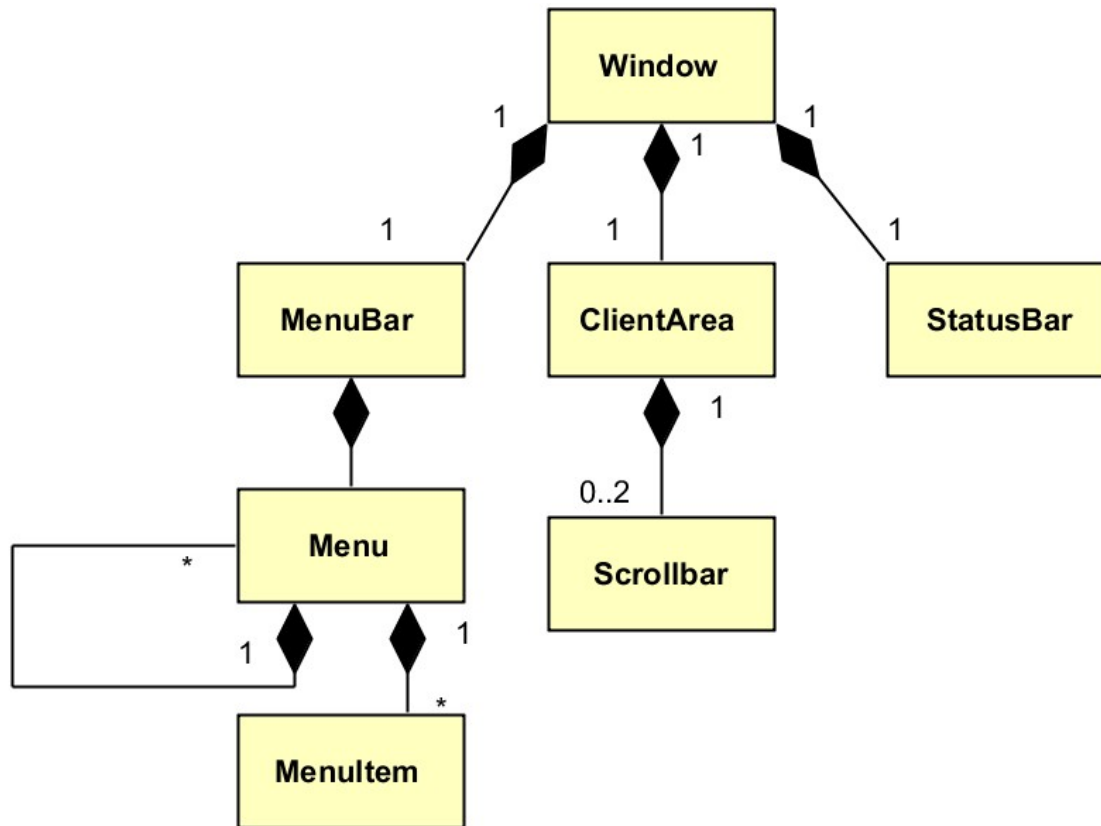
- **Single responsibility principle** - a class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.
- **Open–closed principle** - software entities should be open for extension, but closed for modification.
- **Liskov substitution principle** - Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- **Interface segregation principle** - Many client-specific interfaces are better than one general-purpose interface.
- **Dependency inversion principle** - depend upon abstractions, not concretions.

# Class Hierarchy





# Object Hierarchy



# Object-Oriented Approach to Programming

Additional elements of the object model [Booch]:

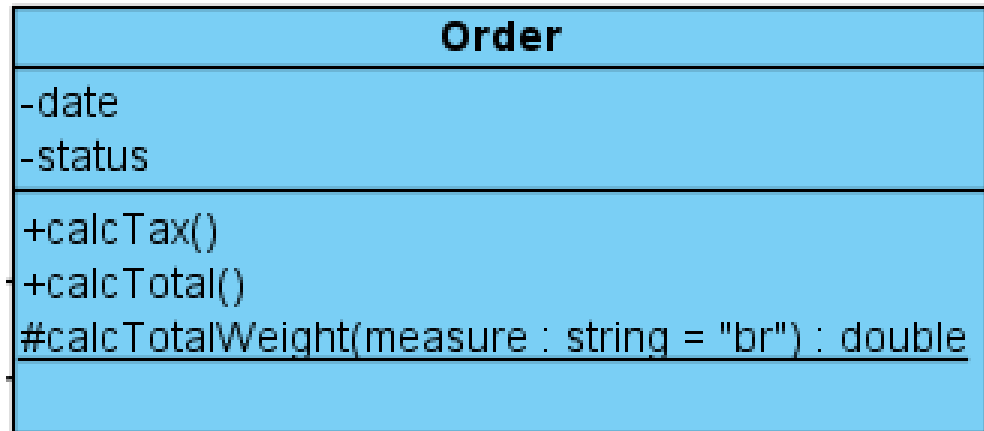
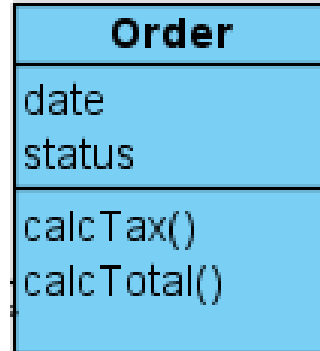
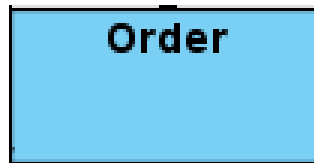
- ❖ **typing** – requirement for the class of an object such that objects of different types can not be replaced (or can in a strictly limited way)
  - static and dynamic binding
  - polymorphism
- ❖ **concurrency** – abstraction and synchronization of processes
- ❖ **length of life** – object-oriented databases

# Classes

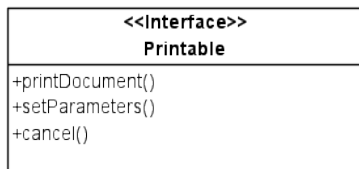
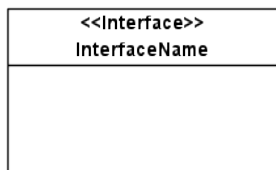
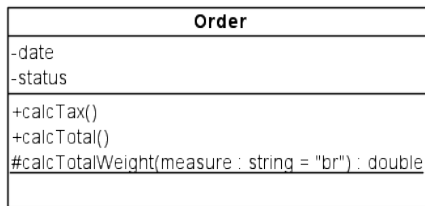
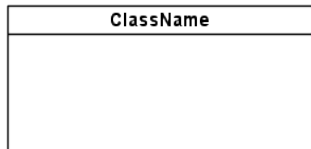
**Class** – describes a set of objects that share the same specifications of the characteristics (attributes and methods), constraints and semantics

- attributes – instances of properties in UML, they can provide end of association, object *structure*
- operations - behavioral characteristics of a classifier, specifying name, type, parameters and constraints for invoking definitely associated with the operation behavior

# Classes - Graphical Notation in UML

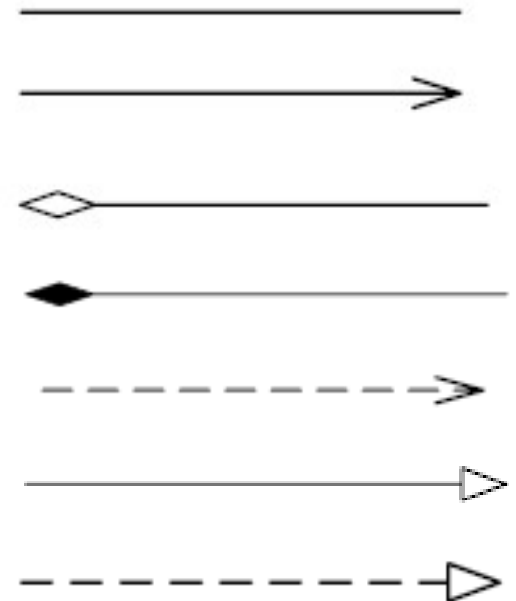


# Elements of Class Diagrams



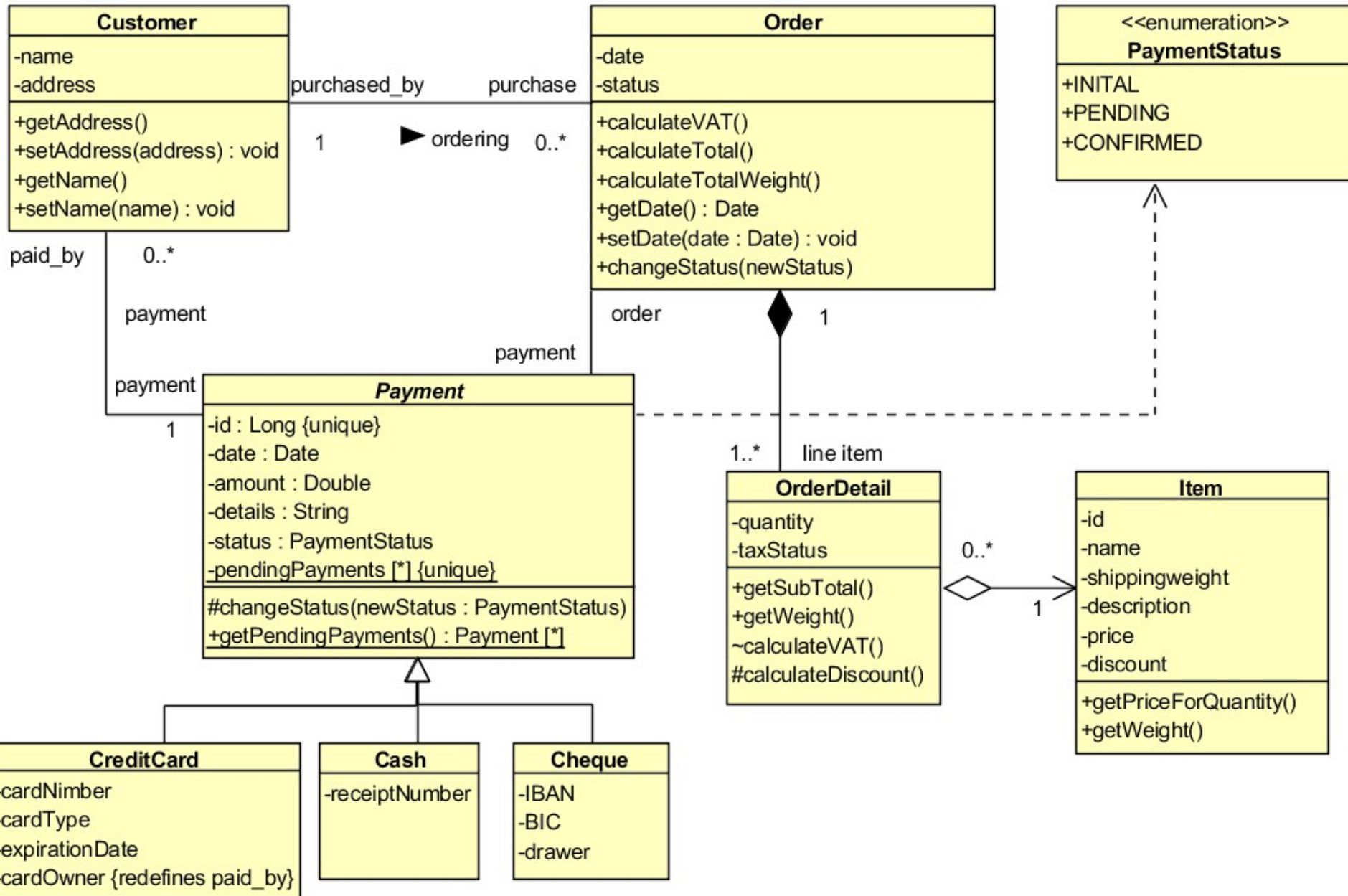
## Types of connections:

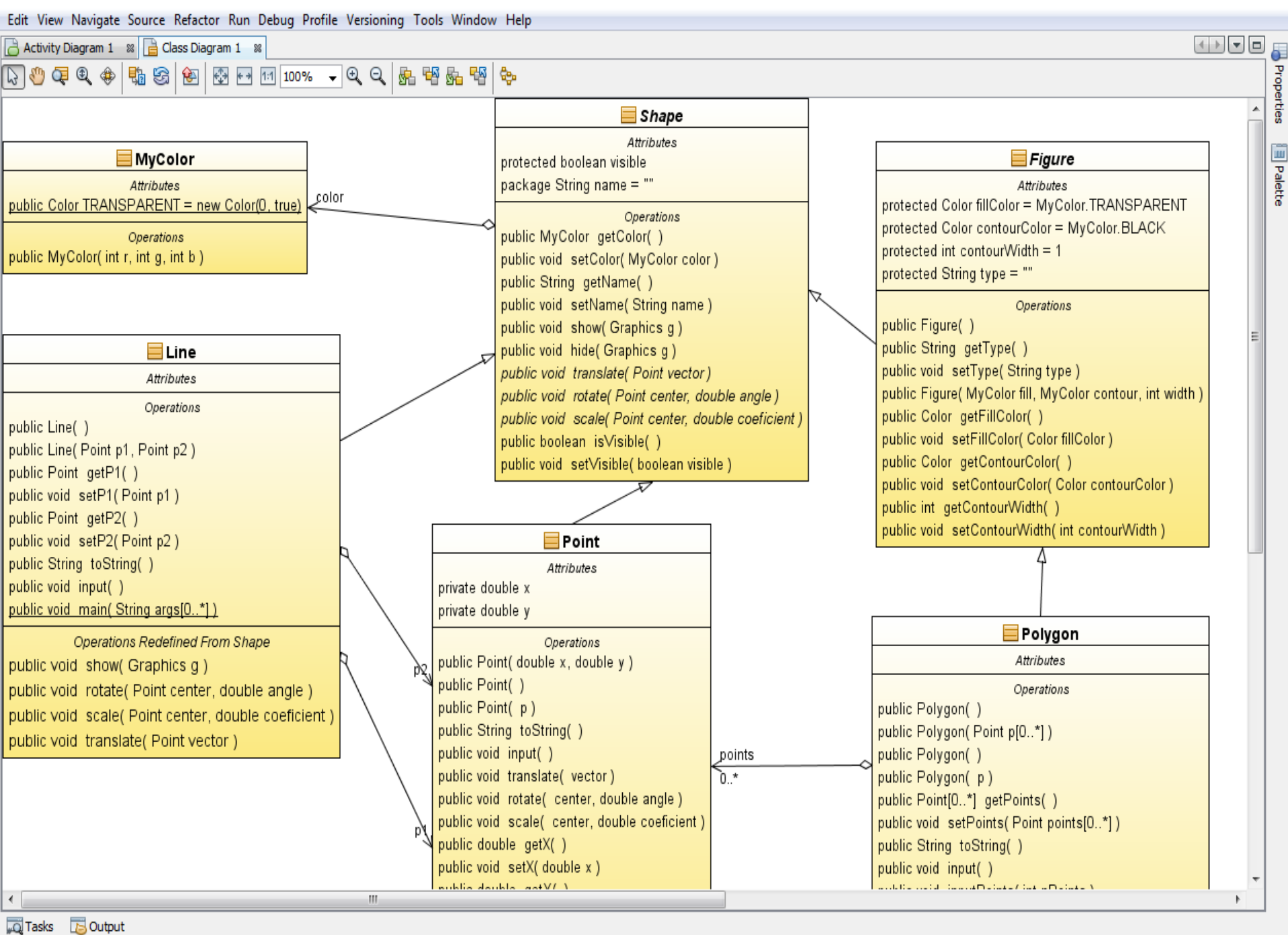
- association
- aggregation
- composition
- dependence
- generalization
- realization





# Class Diagram - 1





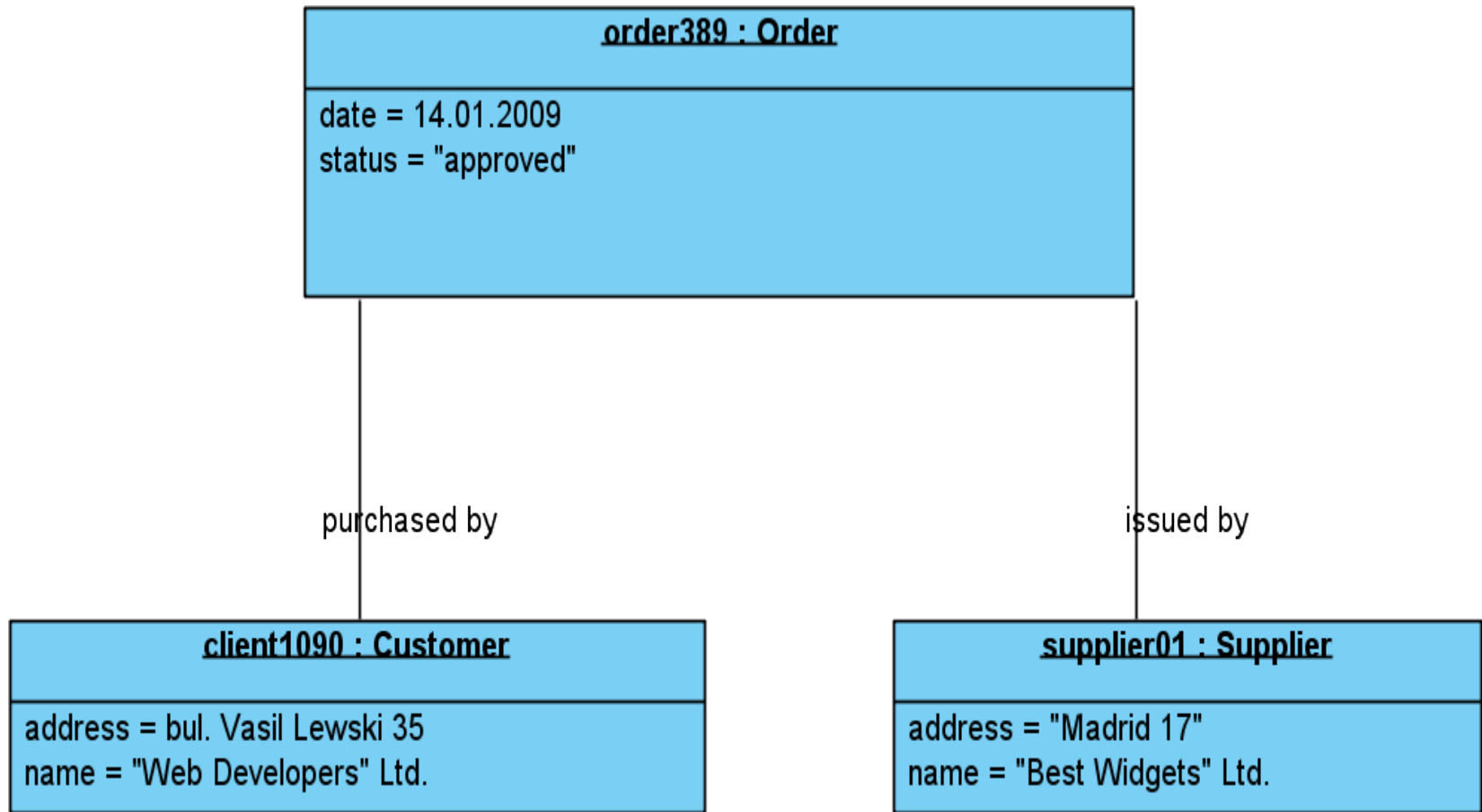
# Objects

**Instance specification = Object** – represents an instance of the modeled system, for example class -> object association -> link, property -> attribute, etc.

- can provide illustration or example of object
- describes the object in a particular moment of time
- may be uncomplete
- Example:

<b>order389 : Order</b>
date = 14.01.2009 status = "approved"

# Object Diagram



# Analysis Classes Stereotypes

Analysis classes are used in the mapping and analysis of system architecture - they present rather different roles and responsibilities, than specific classes to be realized, and are independent of implementation technology:

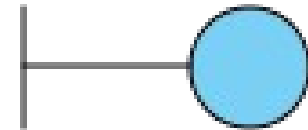
- <<controll>> - business logic
- <<entity>> - data
- <<boundary>> - user or system interface



Controlling Class



Class Unit



Border Class



# Reusing Classes

- ❖ Advantages of code reuse
- ❖ Ways of implementation:
  - Objects composition
  - Inheritance of classes (object types)
- ❖ Building complex objects by composition
- ❖ Initializing the references:
  - on declaration of the site
  - in the constructor
  - before using (lazy initialization)

# Class Inheritance - I

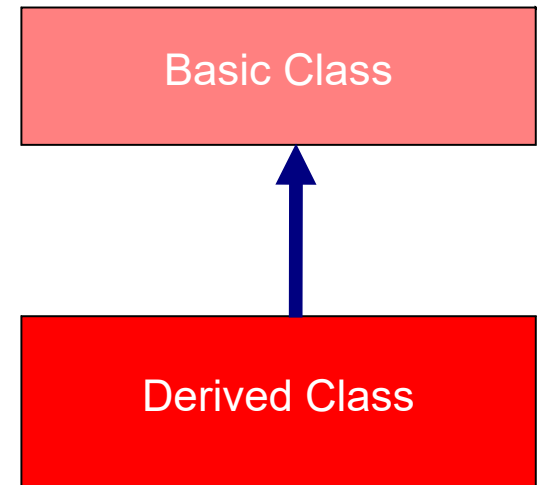
- ❖ Inheritance realization in Java™ language
  - Keyword **extends**
  - Keyword **super**
- ❖ Initialization of objects inheritance:
  - 1) base class; 2) inherited class
    - Calling the default constructors
    - Calling constructors with arguments
- ❖ Combining composition and inheritance

# Class Inheritance - II

- ❖ Clearing of objects – realization in Java™
- ❖ Overloading and overriding methods of base class in derived classes
- ❖ When to use composition and when inheritance?
  - Do we need the interface of the base class?
  - Connection Type - „there is“ and „it is“?

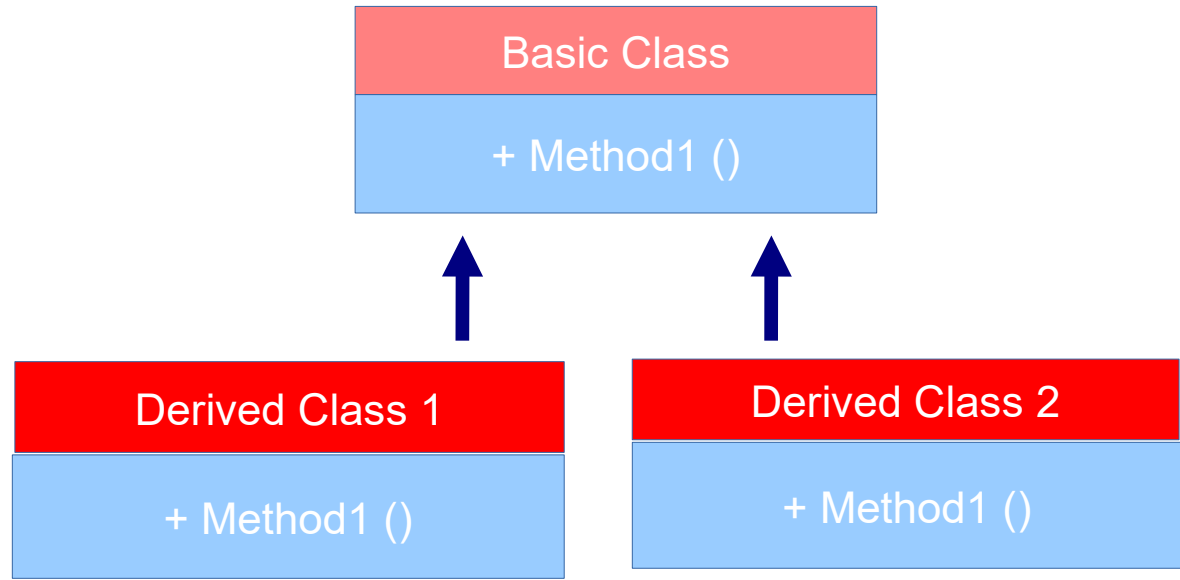
# Class Inheritance - III

- ❖ Protected methods
- ❖ Upcasting
- ❖ Keyword final
  - Final data – defining constants
    - simple data type
    - objects
    - empty fields
    - arguments
  - Final methods
  - Final classes



# Polymorphism - I

## ❖ Upcasting



- ❖ Abstract methods and classes – abstract
- ❖ Order of constructor calls
- ❖ Inheritance and expansion



# Polymorphism - II

- ❖ **Polymorphism** – by default, unless the method is declared as static or final (private methods become automatically final)
- ❖ When constructing objects with inheritance each object cares about its attributes and **delegate initialization of parental attributes on parental constructor or method**
- ❖ Using polymorphic methods in constructor
- ❖ **Covariance** types of return (from Java SE 5)
- ❖ Composition <-> Inheritance - **State Design Pattern**

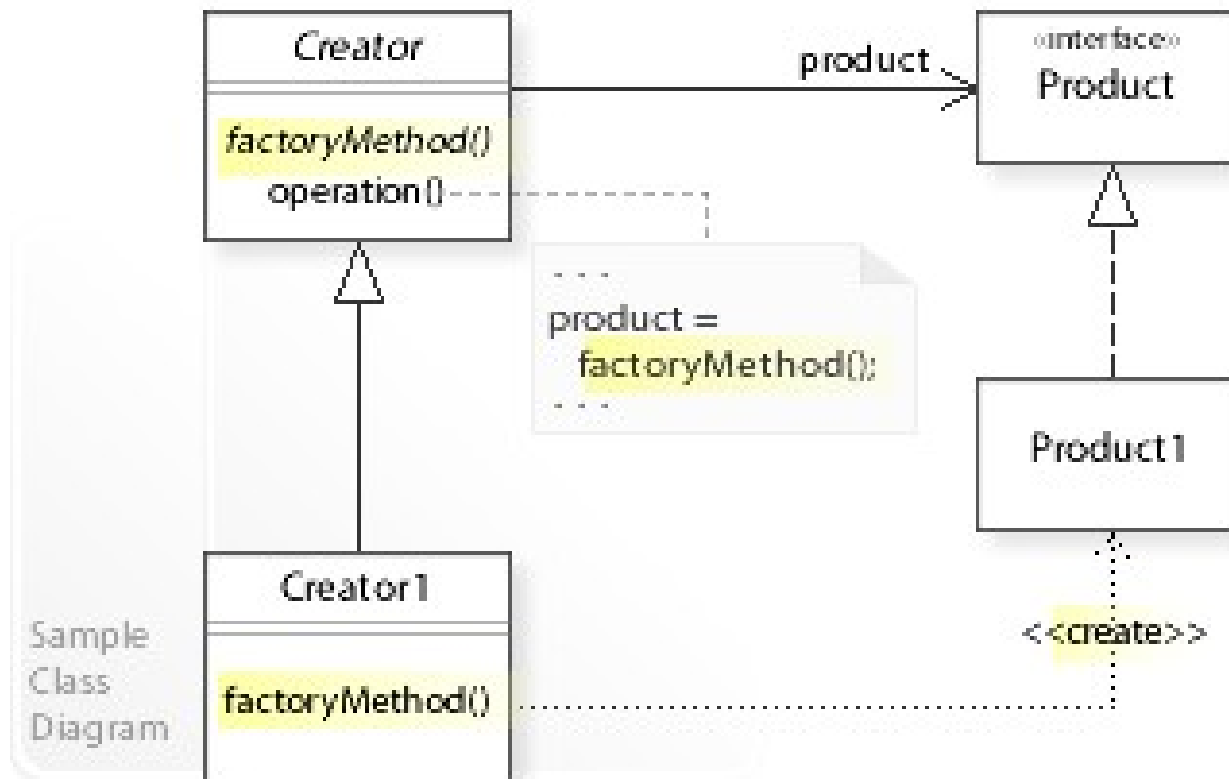
# Interfaces and Multiple Inheritance

- ❖ Interfaces – keywords: **interface**, **implements**
- ❖ Multiple inheritance in Java
- ❖ Interface expansion through inheritance
- ❖ Constants (static final)
- ❖ Interface incorporation

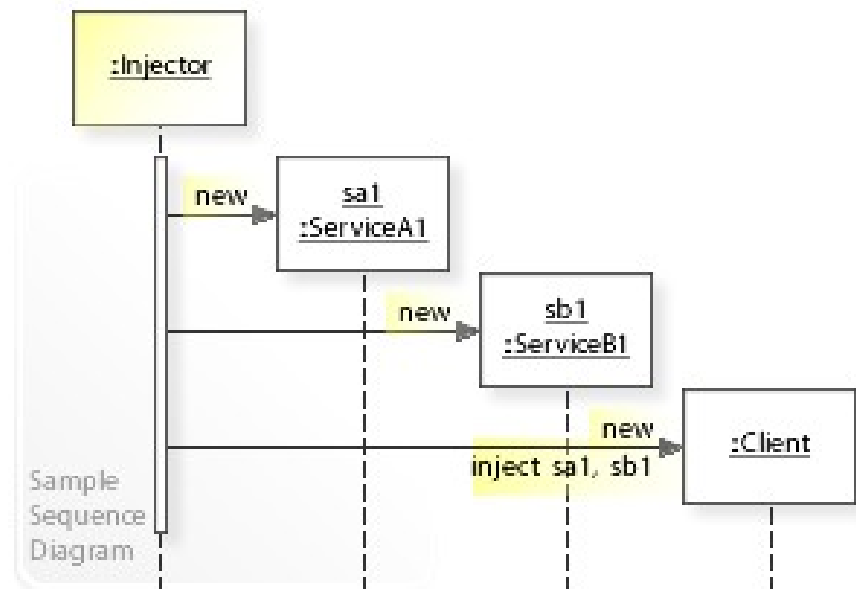
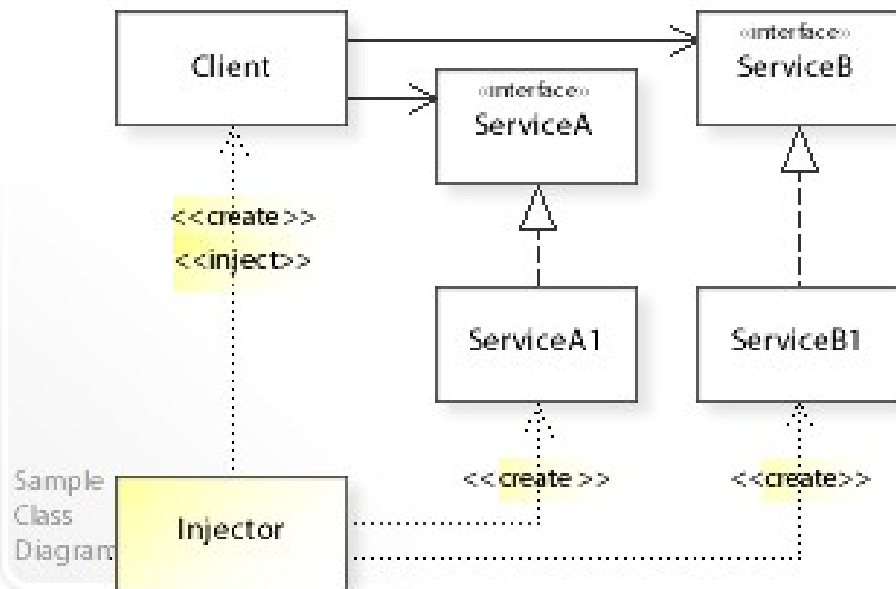
# Advantages of Using Interfaces

- ❖ **Interfaces** cleanly separate requirements type of the object from many possible implementations and make our code more universal and usable
- ❖ **Reusable Design Pattern: Adapter** – It allows to adapt existing realization interface that is required in our application
- ❖ **Inheritance (expansion) of interfaces**
- ❖ **Reusable Design Pattern: Factory Method** – creating reusable client code, isolated from the specifics of the particular server implementation

# Factory Method Design Pattern

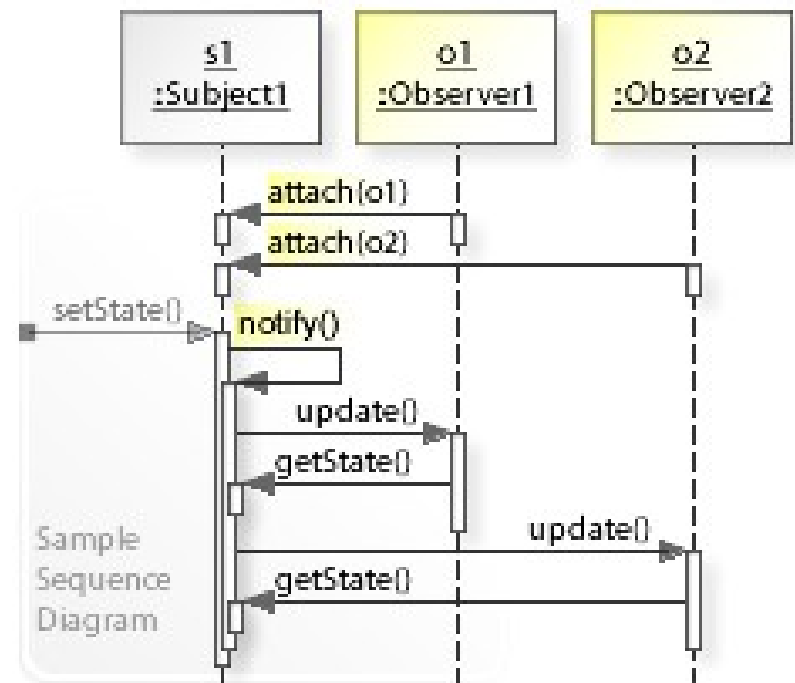
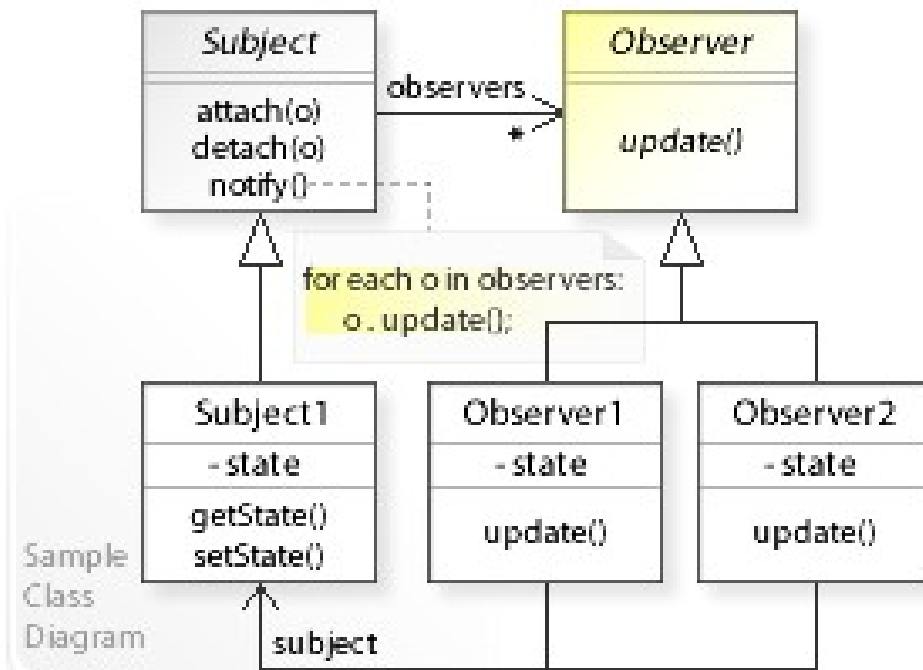


# Dependency Injection Design Pattern

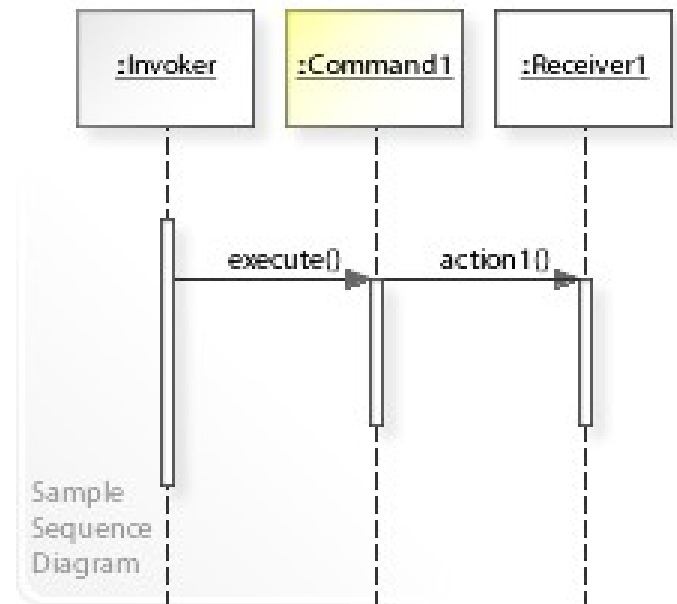
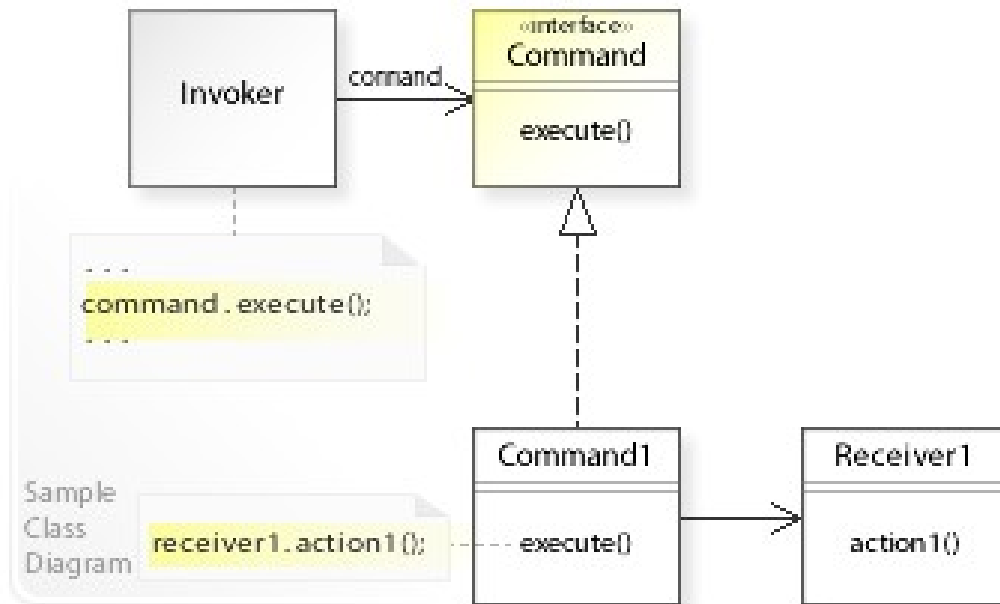




# Observer Design Pattern



# Command Design Pattern



# Inner Classes - I

- ❖ **Inner Classes** group logically related classes and control their visibility
- ❖ **Closures** – internal class has a constant connection to containing outside class and can access all its attributes and even final arguments and local variables (if defined in the method or block)
- ❖ Inner classes can be **anonymous** if used once in the program. Construction.
- ❖ Reference to the object from an external class - **.this** and creating an object from internal class in the context of containing object of the outer class - **.new**

# Inner Classes - II

## ❖ Inner Classes

- defined in an external class
- defined in method
- defined in a block of operators
- access to the attributes of the outer class and to the arguments of the method which are defined in

## ❖ Anonymous inner classes

- realizing public interface
- inheriting class
- instance initialization
- static inner classes

# Enumeration Types

```
public class MyEnumeration {  
    public enum InvoiceType { SIMPLE, VAT }  
    public static void main(String[] args) {  
        for(InvoiceType it : InvoiceType.values())  
            System.out.println(it);  
    }  
}
```

Результат: SIMPLE  
VAT

# Обработка на изключения в Java

- Задължителна обработка на изключенията в езика Java → сигурен и надежден код
- Разделяне на бизнес логиката на програмата от кода за обработка на грешки
- Клас **Throwable** → класове **Error** и **Exception**
- Генериране на изключения – ключова дума **throw**
- Обработка на изключения:
  - **try – catch – finally** блок
  - прехвърляне към извикващия метод - **throws**



# Try-Catch-Finally Block

❖ Оператор **try** за изпълнение на несигурен код, множество **catch** блокове за обработка на изключения и **finally** за гарантиран clean-up накрая на обработката:

```
try {  
    //код, който може да генерира изключения Ex1, Ex2, ...  
} catch(Ex1 ex) { // изпълнява се само при Ex1  
    //вземаме подходящи мерки за разрешаване на проблем 1  
} catch(Ex2 ex) { // изпълнява се само при Ex2  
    //вземаме подходящи мерки за разрешаване на проблем 2  
} finally {  
    //изпълнява се винаги, независимо дали има изключение  
}
```

# Обработка на изключения в Java - 2

- Реализация на собствени изключения
- Конструктори с допълнителни аргументи
- Влагане и повторно генериране на изключения – причина Cause
- Специфика при обработката на **RuntimeException** и неговите наследници
- Завършване чрез **finally**

# Новости при обработката на изключения в Java 7

- Обработка на множество изключения от в една и съща **catch** клауза:

```
catch (Exception1|Exception2 ex) {  
    ex.printStackTrace();  
}
```

- Програмен блок **try-with-resources**:

```
String readInvoiceNumber(String myfile) throws IOException {  
    try (BufferedReader input = new BufferedReader(new  
        FileReader(myfile))) {  
        return input.readLine();  
    }  
}
```

# Thank's for Your Attention!



**Trayan Iliev**

**CEO of IPT – Intellectual Products  
& Technologies**

<http://iproduct.org/>

<http://robolearn.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

<https://www.facebook.com/IPT.EACAD>

<https://plus.google.com/+IproductOrg>